IDA PAPER P-2061

RECOMMENDED SOFTWARE STANDARDS FOR USE BY THE DEFENSE LOGISTICS AGENCY

David Carney

May 1988

Prepared for
Defense Logistics Agency

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, or (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Papers

Papers normally address relatively restricted technical or policy issues. They communicate the results of special analyses, interim reports or phases of a task, ad hoc or quick reaction work. Papers are reviewed to ensure that they meet standards similar to those expected of refereed papers in professional journals.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts to record substantive work done in quick reaction studies and major interactive technical support activities; to make available preliminary and tentative results of analyses or of working group and panel activities; to forward information that is essentially unanalyzed and unevaluated; or to make a record of conferences, meetings, or briefings, or of data developed in the course of an investigation. Review of Memorandum Reports is suited to their content and intended use.

The results of IDA work are also conveyed by briefings and informal memoranda to sponsors and others designated by the sponsors, when appropriate.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release/unlimited distribution.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Paper P-2061	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Institute for Defense Analyses 6b OFFICE SYMBOL: IDA	7a NAME OF MONITORING ORGANIZATION OUSDA, DIMO
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311
8a NAME OF FUNDING/SPONSORING ORGANIZATION Defense Logistics Agency 8b OFFICE SYMBOL (If applicable) DLA	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) DLA-ZWS, 3A636 Cameron Station, Alexandria, VA 22304-6100	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-T5-423 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Recommended Software Standards for Use by the Defense Logistics Agency (U)	
12 PERSONAL AUTHOR(S) David Carney	
13a TYPE OF REPORT: Final 13b TIME COVERED: FROM _____ TO _____	14 DATE OF REPORT (Year, Month, Day): 1988 May 15 PAGE COUNT: 42
16 SUPPLEMENTARY NOTATION	

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software standards; Ada programming language; software configuration management (SCM); portability; coding standards; prototyping.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this IDA Paper is to recommend a set of software standards for use by the Defense Logistics Agency. These recommendations are related to the effort by the DLA to evaluate the Ada progamming language as an Agency standard. The recommendations cover three areas of software development: software configuration management (SCM), portability of Ada programs, and Ada coding standards. Although the principal scope of this paper is that of software standards and their potential use by the DLA, many of its recommendations, principally those concerning coding standards and portability, are derived from lessons learned during the earliest stages of the DLA Ada Prototype Project, which is described in Section 3.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Audrey A. Hook	22b TELEPHONE (Include area code) (703) 824-5501	22c OFFICE SYMBOL IDA/CSED

IDA PAPER P-2061

RECOMMENDED SOFTWARE STANDARDS FOR USE BY THE DEFENSE LOGISTICS AGENCY

David Carney

May 1988

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-T5-423

CONTENTS

PREFACE

The purpose of IDA Paper P-2061, *Recommended Software Standards for Use by the Defense Logistics Agency*, is to make available the findings and conclusions reached during the preliminary stages of IDA task T-T5-423, Defense Logistics Information System. These findings and conclusions take the form of recommendations in three areas of software development: software configuration management (SCM), portability, and coding standards.

Reviewers of this document included: Bill Brykczynski, Audrey Hook, Joseph Linn, Katydean Price, Robert Winner, and James Wolfe.

1. INTRODUCTION

1.1 PURPOSE

The purpose of this paper is to recommend a set of software standards for use by the Defense Logistics Agency. These recommendations are related to the effort by DLA to evaluate the Ada programming language as an Agency standard.

1.2 SCOPE

These recommendations cover three areas of software development: software configuration management (SCM), portability of Ada programs, and Ada coding standards. Recommendations in the first area are independent of any particular programming language. The second area is one in which few standards have yet found widespread acceptance, and the recommendations are only of a very general nature. The third area, Ada coding standards, includes such issues as naming conventions, appropriate and inappropriate statement types, and packaging conventions.

Although the principal scope of this paper is that of software standards and their potential use by DLA, many of its recommendations, principally those concerning coding standards and portability, are derived from lessons learned during the earliest stages of the DLA Ada Prototype Project described in Section 3. Among these experiences were the attempt to erect a workbench of Ada tools and the on-going experiences gained from several intensive Ada training sessions. The workbench experience is documented in IDA Memorandum M-387, *Compiling and Porting the NOSC Tools for Use by the Defense Logistics Agency.*

1.3 BACKGROUND

The Defense Logistics Agency is engaged in a long-term program, the Logistics Systems Modernization Program (LSMP), to modernize its Automated Information Systems. A principal thrust of the LSMP is to determine the feasibility of using the Ada language in DLA applications.

To that end, IDA and DLA are engaged in an Ada Prototype Project. This project involves the creation of a preliminary Ada Programming Support Environment, training a group of DLA programmers in the use of Ada, and the writing of large-scale software projects to demonstrate Ada's capabilities. The functional nature of these demonstrations will be similar to that of the COBOL software currently in use at DLA. Since the project will be distributed through three different machine tiers, an IBM, a Gould, and several Zenith PCs, it will also demonstrate Ada's capacity for portability of software.

One principal requirement for the Prototype Project is an Ada Programming Support Environment (APSE). The APSE is a "workbench" of Ada tools which can later be expanded and modified as the modernization project itself is expanded and modified.

In IDA Memorandum Report M-294, *Ada Prototype Project*, it was recommended that tools for the APSE be acquired by first examining public domain software before resorting to commercially available software. A major source of public domain software is the SIMTEL20 Repository, which contains a large number of available Ada programs. Many of these were selected for potential inclusion in the DLA APSE. Since these tools were commissioned by the Naval Ocean Systems Center, they are commonly referred to as the NOSC tools.

A team of DLA personnel was selected for training in Ada. Their collective expertise is in COBOL, C, and Pascal, and most of the members are proficient in traditional Automated Data Processing methodology. In addition, two of the team members have considerable backgrounds in systems programming.

The team received six weeks of intensive training in Ada. The classes were given by personnel from TeleSoft, Inc., whose compiler will be used during the Prototype Project and beyond. The textbooks for this course were:

- *Software Engineering with Ada*, Booch
- *Reusable Software Components with Ada*, Booch
- *Understanding Concurrency in Ada*, Kenneth Shumate

1.3 APPROACH

There are three different categories of recommendations in this paper, and they derive from numerous sources:

- The trials of erecting the APSE, and the insurmountable difficulty of porting much of it to DLA, were the sources of many of the recommendations concerning configuration management and portability.
- Observing the DLA team's training sessions provided valuable insight into the the type of coding standards and guidelines needed. In addition, many experiences of the authors of some NOSC tools provided other coding recommendations.
- Finally, general experiences gained in using other Ada environments contributed to some of the recommendations in all categories.

Section Two of this paper presents findings and conclusions based on these experiences.

Section Three presents recommendations and provides a rationale for each recommendation.

2. FINDINGS AND CONCLUSIONS

Findings in the three areas of SCM, portability, and coding standards are discussed, and conclusions in each area are presented.

2.1 SOFTWARE CONFIGURATION MANAGEMENT

Finding 1. Many of the NOSC tools chosen for the DLA APSE exist in multiple and incompatible versions. This principally applies to the large tools.

Finding 2. The software environment at DLA presently has few tools to aid or enforce SCM. Erection of a dependable system will substantially impact the success of the entire Ada prototype project.

Finding 3. Major components of the SCM system at DLA will need to include controls for multiple versions of source files, mechanisms for standardized software releases, and management of released object code.

2.1.1 Discussion

Of the problems encountered in compiling the NOSC tools, those stemming from poor SCM were the most difficult to solve. Principally, there were variant versions of many packages, due to the fact that the SIMTEL20 sources reflect multiple releases of the tools. The variants existed at different levels of software: some packages had been subsumed into other packages, creating inconsistent dependencies. Others merely reflected earlier and later versions of the same package. These difficulties were especially prominent in large tools comprising many source files.

These problems were overcome with difficulty. It is noteworthy that IDA also had available another version of some NOSC tools, obtained through GTE. While this other

source of the tools added some confusion, it was only from the GTE versions that the needed versions of some packages were found.

2.1.2 Conclusions

For a software project of any substance, there is need for a dependable SCM system. Given the nature of Ada, where a basic intention is to achieve a high degree of modularity, effective SCM is even more crucial. An acceptable SCM system will minimally contain a set of protocols and standards for version and revision control, as well as a means to map the correct version of source code to the object code. An acceptable SCM system will also contain a reliable mechanism for tracking and documenting releases.

Additionally, an Ada project will need a mechanism that governs compilation of several files. This entails determining the correct order of compilation as well as performing the actual invocation of the compiler.

2.2 PORTABILITY

Finding 4. The NOSC toolset contains several tools that were written with non-validated compilers. These tools will not compile with a valid compiler.

Finding 5. The majority of the tools examined contained system-dependent code. In the larger tools, the dependencies were often dispersed through several packages, making them highly difficult to port to other systems.

Finding 6. Even with these dependencies removed, the large NOSC tools could not be compiled using the Gould compiler at DLA. One of the tools was eventually compiled by a considerable reworking of the sources. The executable that was created failed in elaboration.

2.2.1 Discussion

Of the PC-based tools examined, half have been successfully ported to DLA; of the mainframe-sized tools, none. In the case of the failing PC-based tools a major cause was poor or illegal code, due to use of non-validated compilers as development environments. In the case of the mainframe-sized tools, it was due partially to overreliance on VAX† system calls, and especially to a capacity limitation of the DLA Gould compiler. The attempt to compile the NOSC tools is discussed in detail in IDA Memorandum Report M-387, *Compiling and Porting the NOSC Tools for use by the Defense Logistics Agency*.

2.2.2 Conclusions

It is generally misleading to speak of truly "portable" code; such software is relatively rare. The term "portable" more often refers to code that needs only a small amount of alteration in order to compile on different machines; portability is thus a measurement of such alteration. Code which has a high degree of is a portability is code wherein the needed alterations are easily made. Conversely, non-portable code requires complicated alterations, or is written in such a way that the location where alterations are needed is not easily determined.

It is also apparent that portability may be an issue over which the programmer has little or no control. In the case of the NOSC tools, even though the Gould computer is a reasonably large machine, its compiler was not able to compile code that had compiled without problem on a VAX.

† VAX is a registered trademark of Digital Equipment Corporation.

Since one of the stated goals in the DLA Ada project is achieving software portability through three machine tiers, this issue is fundamental; it is one of the prime points that the project is meant to demonstrate. The DLA team must obviously regard portability as an element that must be present at the earliest level of design, and not, as in the NOSC tools, as a consideration after the code has been written. They should also be aware of the limitations of their compiler, and should be urged to design their code accordingly.

2.3 CODING STANDARDS

Finding 7. The coding standard found in the course's textbooks represents only one possible standard for good Ada code.

2.3.1 Discussion

Most of the DLA team members came to Ada from backgrounds in COBOL, and have need of guidelines in writing Ada code. While the definition of an "Ada style" is, to some extent, a matter of opinion, there is is a genuine need for some practical guidance in this area.

At the present, the primary models available to the team members derive from the texts used in the training sessions. In addition, the code for the NOSC tools provide a wide range of code quality and conventions. The code in the textbooks represents a particular kind of coding style. This style is discussed in detail in Section Three. The code in the NOSC tools ranged from unacceptable to excellent; the experiences of some of the NOSC tool authors provided a source for some of the recommendations made later in this paper.

2.3.2 Conclusions

One of the fundamental aims of the Ada language is the writing of maintainable,

reusable code. As a means toward that end, it is vital that code be easily readable: in one sense, readability is the hallmark of well-crafted code. Readability in this sense does not refer to documentation, but rather to the actual compilable code. In particular, coding practices that favor dense, abbreviated code are to be avoided.

Coding conventions are, in themselves, major contributors to code readability. Also, since Ada usually involves simultaneous references to multiple source files, any coding conventions that simplify these references are beneficial; any that hamper it are poor.

3. RECOMMENDATIONS

There are three categories of recommendations: software configuration management (SCM), portability, and coding standards. Five recommendations for SCM, four recommendations for portability, and fifteen recommendations for coding standards are presented.

3.1 SOFTWARE CONFIGURATION MANAGEMENT

1. *Since the Gould computer will be the major development area, configuration management protocols should be governed by the UNIX® operating system.*

   The eventual disposition of the Ada project through the three architectural tiers is not yet determined. It is clear, however, that the Gould, using the UNIX operating system, will be central to the project. UNIX also provides a foundation of language-independent tools that partially offset the current absence of a true APSE.

2. *The DLA team should be encouraged to use available SCM tools at every stage of the Ada project.*

   There are existing tools that have proven beneficial to SCM. UNIX's *make* and RCS utilities are examples of them. (*make* is a tool that automates compilation of large systems, and RCS is a revision control system for controlling changes to text files.) In addition to encouraging use of tools such as these, other tools, such as mechanisms that automatically generate makefiles, or that facilitate using RCS, should also be developed. Implementation of these mechanisms has already begun.

3. *A mechanism to permit orderly release of source files and executables should be developed.*

   The notion of "releasing" software is present when there are several programmers working on interrelated code. There must be an orderly process that allows tested software to be

used by others, but that permits a programmer continually to improve it. Such a process depends on many factors: the released version of the source must be accessible; the compiled code must be stored in a safe location, so that other users who rely on it can do so indefinitely; and the mechanism whereby a release is made must be easily invoked so that it will be used often.

4. *A mechanism that tracks and documents releases should be developed.*

   The need for tracking releases is vital. It is often necessary, for instance, to rescind an erroneous release, a process that involves reconstruction of an earlier configuration of the system. Without a tracking mechanism, reconstruction of any particular system configuration is likely to be impossible.

5. *SCM standards for the DLA project should be adhered to by all team members without exception.*

   Though notional agreement with this recommendation is probably near universal, experience has shown that SCM standards are those that are followed the least. Experience has also shown that lapses in this area are the most damaging. There are, for instance, costs that can propagate far beyond the awareness of the developers. This point is amply demonstrated by the NOSC tool experience.

   One guideline for DLA in selecting its SCM standards is the ANSI/IEEE Std 828-1983, *Software Configuration Management Plans*. It is recommended that the DLA team investigate this document before making any specific decisions in the area of SCM.

3.2 PORTABILITY

1. *For each given program, all system interface should be isolated in a single package.*

Software must communicate to the native operating system. In the case of code written for the DEC Ada compiler, for instance, system calls are invoked through a package STARLET, supplied by DEC. On UNIX systems, Pragma Interface(C) or Pragma Interface (UNIX) perform similar roles.

Making such software portable involves isolating this communication in a single location. If the system-dependent code is distributed throughout several packages, then the code will port to another machine only with difficulty.

When the software under consideration involves two or more executables, it is further recommended that each have a separate interface package. This will help avoid a situation encountered in the NOSC tools, where a system interface package was used by several executables. The interface was changed for some, but not all of the executables, resulting in an untenable set of package dependencies. Using a separate package for each executable ensures a necessary independence of executable programs.

2. *Reliance on constants that are system-dependent should be avoided.*

Constants such as those found in package System concern capacities of the host compiler, such as the degree of precision in real numbers. If code depends on a factor like this, then that code is not really portable.

The following code will compile with the DEC Ada compiler, but will fail with some others:

```
package Real_Numbers is
...
type Big is digits 15;
                -- This will fail unless the
                -- value of System.Max_Digits
                -- is at least 15
....
```

One possible alternative is:

```
with System;
package Real_Numbers is
....
type Big is digits System.Max_Digits;
....
```

But this solution is invalid if there is genuine need for the greater precision. In that case, however, the code will always be erroneous on a smaller machine, and is not portable at all.

By contrast, the following code *involves* constants from package System, but does not *depend* on any particular values for them:

```
with Text_IO;
procedure Numbers is
  package TIO renames Text_IO;
    max_size : constant Integer := (Integer'width) - 2;
    number   : Integer;
    dummy    : String (1 .. 100);
    len      : Natural;

                -- this code will work regardless of
                -- the actual size of Max_Int.
begin
  TIO.Get_Line (dummy, len);
  if len <= max_size then
    number := Integer(dummy(1..len));
  else
    TIO.Put_line ("Input value too large");
  ...
```

3. *Excepting generics, source files should contain a single compilation unit. In the case of generics, source files should contain precisely one generic specification and one generic body.*

When a source file contains more than one compilation unit and one of the units fails in

compilation, different compilers will behave differently. One possibility is for the compiler to reject the entire compilation; that strategy is used by the Gould compiler. If the source file is very long, with numerous compilation units, and the failure occurs at the very end of the compilation, the wasted time can be considerable.

As a single exception to this recommendation, the Ada Language Reference Manual (ANSI/MIL-STD 1815A) permits an implementation to require generic specifications and bodies to share the same source file. Since the Gould compiler makes this requirement, then generic compilation units should be the only occasion when one source file contains more than one compilation unit. In such cases, the source file should contain no more than two units.

4. *Large arrays, those larger than 1000 elements, should be initialized by slice assignments and not by a single aggregate assignment. If possible, such data structures should be avoided.*

This recommendation stems from the principal reason that the large NOSC tools could not successfully compile at DLA. Several packages in the tools were automatically generated code, containing large aggregates of integers. These aggregates were initialized by a single assignment statement. In all cases, these packages failed to compile at DLA.

The solution in this case was to break the large aggregate assignment into smaller slice assignment. Thereafter, one such package was successfully compiled. But the executable that was generated failed, and it is has not been determined whether the tools can be brought to successful execution under any conditions.

It seems a safer course to recommend that the Ada style in packaging, i.e., small, modular packages, be brought to bear on data structures as well. Otherwise, as in the NOSC tools, code can be created which will compile successfully on one compiler and not on another, and there may be no possible way to port it because of capacity limitations.

3.3 CODING STANDARDS

This section contains recommendations about naming conventions, packaging conventions, and other coding conventions.

3.3.1 Naming Conventions

1. *The naming conventions that are established should be consistent throughout the entire project, and used by all members.*

   This point is self-explanatory, but can not be overemphasized. Even if all of the following recommendations are rejected, there is need for consistent naming conventions across the project.

2. *Whenever practical, use descriptive prefixes for subprograms, especially functions.*

   Subprograms are generally entities that "do" things, and the precise nature of what is done should be clear from the subprogram's name. Prefixes like "Is_", "To_", "From_", "Has_" and similar others, can provide this clarity:

```
"Is_"
        - for a function that returns a boolean
        - result from making an identity test.

"Has_"
        - for a function that returns a boolean
        - result from making an attribute test.

"From_",
"To_"
        - for a function that converts a value into
        - another value. The "From" prefix describes
        - the precondition of the function; the "To_"
        - prefix describes the postcondition. The choice
        - is dependent upon the function's principal work.
```

While these prefixes deal with functions only, parallel examples for procedures are easily imagined. As an example of the value of descriptive names, consider the lack of clarity in the following specifications:

```
function Lower_Case (item : Character) return Character;
        — This function returns a lower case character
        — from an upper case character.


function Lower (item : Character) return Boolean;
        — This function returns true if a character is
        — in lower case.
```

These might typically be used as follows:

```
c : Character := 'Z';
begin
...
   if not Lower(c) then      --??lower than what??
      c := Lower_Case (c);
...
```

A preferable, though more verbose, alternative is both self-documenting and consistent:

```
function To_Lower_Case (item : Character) return Character;
function Is_Lower_Case (item : Character) return Boolean;

...
  c : Character := 'Z';
  begin

...

          – conventional use of "is_"
          – indicates a Boolean test
     if not Is_Lower_Case(c) then
        c := To_Lower_Case (c);
...
```

Finally, if the appearance of "... not Is_Lower_Case" is offensive, then the following addition:

```
function Is_Upper_Case (item : Character) return Boolean;
```

leads to:

```
if Is_Upper_Case(c) then
  c := To_Lower_Case (c);
...
```

3. *Abbreviations should not be used in subprogram names. Wherever practical, subprogram names should be entirely self-documenting.*

   The semantic content of abbreviations is a highly subjective matter. While such specifications as:

```
procedure Val (item : Item_Type);
```

   will probably connote "Value" to most people, it is quite possible that

```
function Mat_mpy (mat_1, mat_2 : Mat_Type) return Mat_Type;
```

   will not be meaningful except to its author. Changing this to

```
function Matrix_Multiply (
        matrix_1,
        matrix_2 : Matrix_Type) return Matrix_Type;
```

   results in a considerable increase in readability.

4. *Naming conventions should be chosen so as to avoid unreadable code.*

   There are many viewpoints on good naming conventions, especially as regards names of types and objects. The DLA Ada team used texts by Booch and Shumate. Particularly in

reference to the Booch texts, the DLA team should be made aware of some different points of view.

There are several objections that can be made to the Booch style. First, since all objects begin with the four characters "The_", there is an unwelcome element of sameness to each object. And if there are several objects being manipulated in the code, or several fields of the same record object, the result can be extremely awkward to read. The following is an example:

```
if The_Ring.The_Back = 0 then
  raise Underflow;
elsif The_Ring.The_Back = 1 then
  The_Ring.The_Top := 0;
  The_Ring.The_Back := 0;
  The_Ring.The_Mark := 0;
else
  The_Ring.The_Items(The_Ring.The_Top..The_Ring.The_Back-1):=
  The_Ring.The_Items(The_Ring.The_Top + 1)..The_Ring.The_Back);
  The_Ring.The_Back := The_Ring.The_Back - 1;
  if The_Ring.The_Mark > The_Ring.The_Top then
    The_Ring.The_Mark := The_Ring.The_Mark - 1;
end if;
```

(Booch, p.185)

Another weakness in the above convention is that when two objects of the same type are used, they are distinguished by prepositions, commonly "To" and "From". But the use of these is incorrect as regards common English meaning. As an example:

```
for Index in From_The_Map.The_Items'Range loop
  if From_The_Map.The_Items(Index).The_State = Bound then
    Find (From_The_Map.The_Items(Index).The_Domain,
      To_The_Map, The_Bucket);
```

(Booch, p.230)

The intended meaning here is to distinguish between a "to" map and a "from" map, one a source and one a destination. But in simple English, using "...to the x...from the x..." commonly refers to the same "x". To be consistent with English usage, it would need to read: "The_To_Map ... The_From_Map", at which point common sense rebels.

5. *Wherever practical, use descriptive suffixes to denote common data types.*

It is undoubtedly a good practice to separate type names from variable names; that is an obvious intent of the conventions discussed in Recommendation 4. But a better way to achieve that goal is to place a descriptive suffix on the type, rather than an article on each variable. By using such suffixes as:

"_ptr" – for access types.
"_rec" – for record types.
"_arr" – for array types.
"_type" – for enumeration types.

the following code:

```
type Color_Type is (red, white, blue);
type Color_Ptr is access Color_Type;
color : Color_Ptr;
...
color := new Color_Type'(red);
if color.all /= blue then
...
```

will be both clear and succinct. It should be noted that using abbreviated *suffixes* on types, unlike abbreviations for the *nouns* or *verbs* in subprogram names (cf. Recommendation 3) is an acceptable practice, since suffixes indicate only typical data types such as arrays, records, and pointers.

6. *Use simple names (without prefix or suffix) to denote variables.*

Generally, type names are used once, variables names several times. The descriptive prefix or suffix should be used at the point where the type needs to be discerned, nowhere else. Of the following two examples, the first is preferable:

```
type Node;
type Node_Ptr is access Node;
type Queue_Rec is
record
  Front : Node_Ptr;
  Back  : Node_Ptr;
end record;
...

procedure Copy (
  From : in Queue_Rec;
  To   : in out Queue_Rec) is
...
...
  if From.Front = null then
    To.Front := null;
    To.Back := null;
...
...
```

```
type Node;
type Structure is access Node;
type Queue is
record
  The_Front : Structure;
  The_Back  : Structure;
end record;
...

procedure Copy (
  From_The_Queue : in Queue;
  To_The_Queue   : in out Queue) is
        ...
        ...
         if From_The_Queue.The_Front = null then
           To_The_Queue.The_Front := null;
           To_The_Queue.The_Back := null;
        ...
```

(Booch, p.149)

3.3.2 Packaging Conventions

1. *Subprograms, whether functions or procedure, should be brief; each should accomplish a single action.*

   One of the hallmarks of the Ada style is a high degree of modularization, with the restriction of a subprogram to a single action. The benefits of this are twofold: first, since the subprogram has only one effect, it can subsequently be reused in a variety of contexts. Second, the code of such a subprogram will necessarily prevent the dense, unmanageable code that Ada was intended to avoid.

   As a simple means to achieve this goal, it is further recommended that a typical subprogram be restricted to a very few lines of code. An upward limit is difficult to determine, but a subprogram that is larger than fifty lines is probably too long.

   Some subprograms will perforce violate this recommendation; sometimes such things as a very lengthy case statement are the best solution to a particular problem. But in the general

order, this recommendation can restrict these occasions to a minimum, and can also enforce a logical rigor conducive to good software engineering practice.

2. *Wherever possible, subprograms should have no side effects. All effects of a subprogram should be centered on parameters.*

   The principal way that a subprogram can have a side effect is by acting on global variables. Global variables are generally avoided by modern software engineering practices, and an Ada programming style should generally follow this practice.

3. *The number of parameters for subprograms should rarely if ever exceed six.*

   This is related to recommendation 1 concerning single-action subprograms. If a subprogram genuinely has need of many parameters, it is worth considering whether the chosen data structures are appropriate. A common possibility is that the several parameters can be collected into a single record type, and passed in as a single parameter. If this is not appropriate it is then worth considering if the action of the subprogram is itself appropriate, or whether the subprogram is really doing the work of several procedures.

4. *Wherever possible, variables should not appear in package specifications. Variables whose life span must exceed a given subprogram call should lie in package bodies.*

   The presence of variables in specifications is closely related to recommendation 2 concerning the danger of side effects. A variable in a specification is vulnerable to all units that 'with' the package. The package body is the appropriate location for variables whose life span must exceed a given subprogram call, since only the package's own subprograms may alter such variables.

5. *Constants in package specifications should be replaced by parameterless functions.*

The presence of a constant in a specification is always subject to the danger that the constant will need to be changed and the package recompiled, thus rendering all dependent units obsolete. The effect of a visible constant can be gained without this risk by using a parameterless function to return the constant value. The second version below is preferable to the first:

```
package Data is
 ....
Int_Value : constant Integer := 100;
 ....
```

```
package Data is
 .....
function Int_Value return Integer;
 .....
```

The actual integer value is then located in the package body, which can be altered and recompiled with no other dependencies involved. Note that any calling program that uses this value does so with precisely the same code for both versions:

```
with Data;
procedure Do_Something is
 ...
x := Data.Int_Value;
```

6. *Wherever possible, avoid subunits.*

Most of the asserted benefits of subunits are imaginary. Though textbook examples of development, where a body is stubbed out and the subunits developed one by one, look quite reasonable, experiences by many Ada programmers suggest that such neat sequences

of development seldom occur.

This recommendation is potentially controversial, since textbooks generally urge the frequent use of subunits. But it is the author's experience in various Ada projects that programmers in large numbers come to avoid subunits except in the most exceptional circumstances.

3.3.3 Other Coding Conventions

1. *Avoid unnecessary WITH clauses in specifications.*

   It is not uncommon to include a WITH clause in a package specification even if the 'withed' unit is not referenced until the body. Except for the predefined units such as Text_IO, this practice can have unfortunate results. Principally, it will add unnecessary dependencies, which in turn will trigger unnecessary recompilations throughout the development phase. In addition, such a practice is a mark of poor engineering standards.

2. *Use USE clauses seldom if at all.*

   The principal objection to the USE clause is that it obscures the location of declarations from the reader of the code. Using USE is not the same as 'information hiding': on the contrary, USE hides valuable information from a person who might desperately need the information that is hidden.

   There are only two reasons that USE clauses might be justified:

   - To avoid cumbersome code filled with dot-selected identifiers.
   - To gain visibility of equality and inequality.

   In the first case, it is often a better practice to use package renames, which simplify the appearance of the code and still allow the reader to locate references. As an example, the

second fragment below is preferable to the first:

```
with A_Types, B_Types, C_Types;
use A_Types, B_Types, C_Types;
package Data is
   var_1 : Color := gray;
   var_2 : Shade := Initialize;
   var_3 : Hue := Initialize (var_1);
   var_4 : Hue := Initialize;
```

```
with A_Types, B_Types, C_Types;
package Data is
   package A renames A_Types;
   package B renames B_Types;
   package C renames C_Types;

   var_1 : A.Color := A.gray;
   var_2 : B.Shade := B.Initialize;
   var_3 : C.Hue := C.Initialize (var_1);
   var_4 : C.Hue := B.Initialize;
```

It is also worth noting that the USE version obscures the fact that var_3 and var_4 are initialized by functions in different packages, a point that is explicit in the second version.

The second reason to add a USE clause is to gain the visibility of the equality operator. In such cases, the following are possible alternatives:

a. If the equality visibility is needed only once, then the "=" notation is not a terrible inconvenience.

```
with Data_Types;
package body Something is
  package DT renames Data_Types;
...
  procedure Do_Something is
     x : DT.AnyKind;
  begin
    x := Some_Function;

...
                  – this is the only time
                  – the "=" is needed
              if DT."=" (x,DT.red) then ...
```

b. If the visibility is only needed within a single procedure, then the USE clause can also be located there, as in the following example:

```
with Data_Types;
package body Something is
...
  procedure Do_Something is
    x : Data_Types.AnyKind;

          – AnyKind is defined
          – in package Data_Types
    use Data_Types;
          – USE clause is in effect
          – only within this procedure
    begin
      x := Some_Function;
      if x = red then ...
```

Note also that the USE clause appears only after the declaration of variables: the location of type 'AnyKind' is not hidden by USE.

3. *Exceptions should be used only for true run-time error conditions. They should not be used for recovering from expected conditions.*

In most compiler implementations, exceptions have a high overhead. Further, the intended use of exceptions in the design of Ada was not to include any message-passing functionality,

but only to provide a means to recover from runtime errors.

For instance, consider the following:

```
function Calculate (x : Integer) return Integer is
begin
 .... -- do some useful computation with x
  return x;
exception
  when Constraint_Error => return 10_000_000;
              -- set x to 10_000_000 whenever the
              -- computation exceeds Max_Int
end Calculate;
```

Code such as this is using the exception handling mechanism of Ada to test boundary conditions of the **in** parameter. This is the type of test that might better be made in the code instead, if at all possible:

```
function Calculate (x : Integer) return Integer is
  subtype Acceptable_Range is Integer range <some acceptable range>;

begin
   if not (x in Acceptable_Range) then
         return 10_000_000;
   else
    .... -- do some useful computation with x
   end if;
   return x;
end Calculate;
```

4. SUMMARY

The standards enumerated in this paper are based on lessons learned when COBOL programmers at the Defense Logistics Agency were making a transition to Ada. There is no intention to cover all possible areas, but rather to focus on the standards most commonly needed by experienced programmers making such a transition. These standards should therefore be regarded as a starting point, over which a fuller set of agency-wide standards can be erected. The full complement of DLA software standards can and should be perceived as being a major contribution by the Ada Prototype Project to the eventual success of DLA's Logistics Systems Modernization Plan.

The matter of standards should not be thought of as "elementary," or an issue for novices only. The need for consistent, sensible standards in modern software engineering is indisputable. Especially given the probable scope of projects written in Ada, there can be little doubt that ad hoc, on-the-spot conventions and standards will be detrimental factors in any project's success. From both an engineering and a management viewpoint, the more a project is bound to a uniform, common-sense set of software standards, the more the members of that project are free to focus their energy on the real problems - designs, algorithms, optimizations, abstractions - that face software engineering.

BIBLIOGRAPHY

Gardner, M.R., R.L. Hutchison, & T.P Reagan. "A Portability Study Based on Rehosting WIS Ada Tools to Several Environments." McLean, VA: The MITRE Corporation, 1986. Photocopied.

Nissen, J.C.D. & Peter J.L. Wallis. *Portability and Style in Ada.* Cambridge: Cambridge University Press, 1984.

SofTech, Inc. "Ada Portability Guidelines." Waltham, MA: SofTech, Inc., 1984.

Tracz, Will. "Ada Reusability Efforts: A Survey of the State of the Practice." Stanford, CA: Computer Systems Laboratory, Stanford University, 1987. Photocopied.

Distribution List for IDA Paper P-2061

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Ms. Sally Barnes DLA-ZWS, 3A636 Cameron Station Alexandria, VA 22304-6100	2 copies
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
Mr. Ken Bowles TeleSoft 5959 Cornerstone Court West San Diego, CA 92121-9891	1 copy
Mr. Tim Brass Defense Logistics Agency 3990 East Broad Street Columbus, OH 43216-5002	1 copy
Ms. Tricia Oberndorf Naval Ocean Systems Center Code 425 San Diego, CA 92152-5000	1 copy
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 480 South Aiken Av. Pittsburgh, PA 15231	1 copy
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy
IDA	
General W.Y. Smith, HQ	1 copy
Mr. Philip Major, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Ms. Anne Douville, CSED	1 copy
Mr. Terry Mayfield, CSED	1 copy
Dr. David Carney, CSED	2 copies
Ms. Audrey A. Hook, CSED	1 copy
Mr. Richard Waychoff, CSED	1 copy
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies